FlexBulk: Intelligently Forming Atomic Blocks in Blocked-Execution Multiprocessors to Minimize Squashes

Rishi Agarwal and Josep Torrellas
University of Illinois at Urbana-Champaign, USA
{agarwa29,torrella}@illinois.edu
http://iacoma.cs.uiuc.edu

ABSTRACT

Blocked-execution multiprocessor architectures continuously run atomic blocks of instructions — also called *Chunks*. Such architectures can boost both performance and software productivity, and enable unique compiler optimization opportunities. Unfortunately, they are handicapped in that, if they use large chunks to minimize chunk-commit overhead and to enable more compiler optimization, inter-thread data conflicts may lead to frequent chunk squashes.

In this paper, we present automatic techniques to form chunks in these architectures to minimize the cycles lost to squashes. We start by characterizing the operations that frequently cause squashes. We call them *Squash Hazards*. We then propose squash-removing algorithms tailored to these Squash Hazards. We also describe a software framework called *FlexBulk* that profiles the code and transforms it following these algorithms. We evaluate FlexBulk on 16-threaded PARSEC and SPLASH-2 codes running on a simulated machine. The results show that, with 17,000-instruction chunks, FlexBulk eliminates, on average, over 90% of the squash cycles in the applications. As a result, compared to a baseline execution with 2,000-instruction chunks as in previous work, the applications run on average 1.43x faster.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures - MIMD Processors; D.1.3 [**Programming Techniques**]: Concurrent Programming - Parallel Programming.

General Terms

Design, Performance.

Keywords

Atomic Block Execution, Thread Squash, Speculation.

1. INTRODUCTION

Recent research has outlined a class of shared-memory architectures where processors continuously execute blocks of consecutive instructions from the program (also called *Chunks*) in an atomic manner [2, 6, 7, 9, 11, 18, 21, 22, 23]. These *Blocked-Execution* architectures broadly include TCC [7, 11], Bulk [6, 21], Implicit Transactions [22], ASO [23], InvisiFence [2], DMP [9], and SRC [18]. This execution mode has performance and programmability advantages. For example, it supports transactional memory [7, 11, 18], high performance under strict memory consistency models [2, 6, 21, 23], and several techniques for parallel program development and debugging such as deterministic execution [9], parallel program replay [15], and atomicity violation debugging [12].

Another appealing aspect of these architectures is the performance potential of a compiler that drives the chunk formation. Such a compiler can perform aggressive code transformations inside a chunk that can be *unsafe* under certain conditions — since we know that the hardware guarantees atomic execution or will squash the whole chunk. For example, the compiler can create chunks where code is generated assuming a certain control path is taken [17]. As another example, it can create chunks where code is re-ordered across synchronization operations [1].

A shortcoming of executing in chunks is that chunks might get squashed. Chunks are squashed when they cannot execute atomically. This typically occurs when two concurrently-executing chunks suffer a data conflict — i.e., both chunks access the same memory address and at least one writes. Squashes harm both performance and energy efficiency.

Some previous work indicates that chunk squashes are uncommon in popular applications (e.g., [6]). However, such work uses small chunk sizes of 2,000 dynamic instructions or fewer. In practice, there are two reasons why we want chunks that are several times larger. The first one is to give more room for the compiler to optimize the code within chunks, as suggested above. The second one is to keep overall chunk-commit overhead modest, even in machines with many processors. Unfortunately, large chunks have a higher chance of suffering conflicts (and buffer overflows) that result in squashes.

Given a blocked-execution architecture with large chunks — so that overall chunk-commit overhead is low and future compiler work can perform aggressive optimization — the goal of this paper is to propose automatic techniques to form chunks to minimize the cycles lost to squashes. As most squashes are the result of communication between threads, we must: (i) characterize the communication operations, and (ii) tailor the chunks so that concurrent chunks do not communicate.

Previous work by Ahn *et al* [1] proposed to reduce squashes by tight-fitting chunk boundaries around high-contention critical sections. Unfortunately, our experience shows that most squashes are not induced by a handful of high-contention critical sections. Rather, they are caused by many operations that are spread across the program. We call these operations *Squash Hazards*.

Based on this discussion, this paper makes three contributions. First, we characterize the common types of Squash Hazards in popular applications, and then propose squash-removing algorithms tailored to them. These algorithms consist of simple code transformations, typically embedded inside synchronization macros, that create chunk boundaries for minimal squashes.

Second, we describe a software framework called *FlexBulk* that profiles off-the-shelf multithreaded code for Squash Hazards, and transforms it with the squash-removing algorithms.

ISCA'11, June 4–8, 2011, San Jose, California, USA.

loadtree from Barnes:

```
while(flag){
    :
  if(..) {
      Lock(&CellLock->CL[((cellptr)mynode)->seqnum%MaxLock]);
        :
      Unlock(&CellLock->CL[((cellptr)mynode)->seqnum%MaxLock]);
  }
  if(..) {
      Lock(&CellLock->CL[((cellptr)mynode)->seqnum%MaxLock]);
        :
      Unlock(&CellLock->CL[((cellptr)mynode)->seqnum%MaxLock]);
  }
  if(..) {
      mynode = *qptr;
        :
  }
    :
}
```

Transformations: 1) remove Unlock(expression)
2) replace Lock(expression) by while(expression==taken){ }

(a)

Application		Barnes from SPLASH−2	Database from SpecJVM98
Function		loadtree	shell_sort
Num Loop Iter		18	Blocked to 50
Loop Instructions Before Optimization	Static	879	1,023
	Dyn per iter	517	353
	Dyn per loop	**9,306**	**17,650**
Loop Instructions After Optimization	Static	861	606
	Dyn per iter	432	247
	Dyn per loop	**7,776**	**12,350**
Difference in Iteration Instructions (Dynamic)	Memory	8	59
	Atomics	4	2
	Other	73	45
	Total	85	106

(b)

Figure 1: Examples of compiler optimizations for large chunks. Instruction counts refer to x86 assembly.

Finally, we evaluate FlexBulk on PARSEC and SPLASH-2 codes running on a 16-core simulated blocked-execution architecture. The results show that the techniques are very effective. With 17,000-instruction chunks, FlexBulk eliminates, on average, over 90% of the squash cycles in the applications. As a result, compared to a baseline execution with 2,000-instruction chunks as in previous work, the applications run on average 1.43x faster.

This paper is organized as follows: Sections 2 and 3 give a background and motivation; Section 4 describes the Squash Hazards and squash-removing algorithms; Section 5 describes the FlexBulk framework; Sections 6 and 7 evaluate FlexBulk; and Section 8 discusses related work.

2. BLOCKED EXECUTION

Blocked execution is a mode of execution where a processor continuously executes blocks (or chunks) of consecutive instructions in the program *atomically*. Several shared-memory architectures that operate or can operate in this mode have been recently proposed [2, 6, 7, 9, 11, 18, 21, 22, 23]. In these architectures, before a chunk starts, the processor hardware takes a register checkpoint. Then, the chunk executes speculatively. The architecture keeps a record of the addresses read and written by the chunk, and prevents the written data from being permanently merged with the memory system before the chunk is proven to be safe and commits. At the same time, the architecture watches for data conflicts between the chunk and other concurrently-executing chunks. If a conflict is detected, one of the chunks is squashed and restarted. Squashing involves discarding the data updated by the chunk and restoring the register checkpoint.

This paper uses the Bulk Multicore [6, 21] as an example of blocked-execution architecture. In this architecture, the state generated by a chunk is stored in the cache. The memory addresses read and written by the chunk are hash-encoded in a Read (R) and Write (W) signature register using Bloom filters. Detection of data conflicts between chunks is performed lazily at commit. Specifically, when a chunk ends and wants to commit, the hardware sends the W signature to other, relevant processors. In these processors, the incoming signature is intersected with the local ones to detect conflicts. If a conflict is found, the local chunk is squashed. With this approach, the chunks from all processors appear to execute in a total order.

While a processor is executing a chunk, no other processor can observe the intermediate state of the chunk. Blocked-execution architectures leverage this fact in two ways. First, the hardware can reorder and overlap memory accesses inside chunks, nullifying memory fences [2, 6, 23]. Second, the compiler can perform optimizations inside a chunk that may be unsafe [1, 17]. For example, Neelakantam *et al* [17] generate chunks where code is generated assuming that a certain control path is followed. If another path is taken, the chunk is squashed. Ahn *et al* [1] generate chunks where the code is optimized across synchronization operations, performing register allocation, common subexpression elimination, and other optimizations. If a data collision is detected, the chunk is squashed. If the chunk is squashed by an event that re-appears on re-execution (e.g., cache overflow), execution transfers to a Safe Version of the chunk that is not unsafely optimized.

3. WHY WE NEED LARGE CHUNKS

Some past proposals for blocked-execution multiprocessor architectures have used small chunk sizes of 2,000 instructions or less [6]. However, there are two performance reasons why we need chunks that are several times bigger: to minimize overall chunk-commit overhead and to enable more compiler optimization.

In lazy conflict-detection architectures (the focus of our work), chunk commit is costly. It involves informing other processors of the set of addresses updated by the chunk. Sending out the list of such addresses is expensive. Even if the addresses are encoded in a signature, a machine with large numbers of processors attempting to commit small chunks will surely experience long commit latencies. Past work hid this latency with multiple, pipelined chunks per processor. However, such scheme doubles or quadruples the hardware needed in the processor. Overall, with large chunks, the commit overhead is much less of a worry.

Large chunks also give more flexibility to the compiler to apply new optimizations. While compiler optimization is outside this paper's scope, as a motivation, we consider two code examples with *already existing* optimizations. The first one is the *loadtree* function in *Barnes* from SPLASH-2 (Figure 1(a)). The function has a while loop that traverses a tree. Each iteration has two critical sections with the same lock, but the lock changes across iterations. As conflicts are rare, we perform lock elision [19]. It involves (i) removing the Unlock calls and (ii) replacing the Lock calls by while loops that read the lock variable using *plain* loads until the variable is free [1]. We place the resulting whole loop in a chunk.

Without the atomics, our GCC compiler attempts to perform loop invariant code motion of the lock address. However, since each iter-

ation accesses a different lock, the compiler only moves outside of the loop the generation of the base address of the lock (*&CellLock->CL*), which was computed four times. This saves instructions, including memory accesses. We also observe other improvements from better code and register allocation.

Column 2 of Figure 1(b) shows the impact of the optimization. The loop has 18 iterations. Before the optimization, the code executed 517 instructions per iteration (9,306 per loop); after the optimization, it executes 432 per iteration (7,776 per loop). Each iteration executes 85 fewer instructions, of which 8 are memory accesses and 4 are atomics. Most of these 85 instructions appear outside of the loop and execute only once. These are significant savings. Note that a large chunk that includes all 18 iterations performs the best: if we tile the loop to fit into a smaller chunk, we have to execute the 85-instruction loop startup in every tile.

Column 3 of Figure 1(b) shows a similar optimization in the *Database* code of SpecJVM98. The loop runs for 50 iterations. Before the optimization, the code executed 353 instructions per iteration (17,650 per loop); after the optimization, it executes 247 per iteration (12,350 per loop). Each iteration executes 106 fewer instructions, of which 59 are memory accesses and 2 are atomics. These savings come from optimizing lock address generation, removing synchronization, optimizing null and range checks, and generally optimizing the code and register allocation. Again, a large chunk that includes the whole loop has the least startup cost.

Unfortunately, large chunks have a higher chance of suffering conflicts that produce squashes. To address this problem, this paper proposes user-transparent techniques to intelligently break off-the-shelf multithreaded code into chunks that minimize squashes.

4. ELIMINATING SQUASHES

4.1 Main Idea

Squashes due to data conflicts occur primarily because threads communicate with each other. Consequently, to eliminate squashes, we should first identify the code locations where threads communicate. Then, we should tailor the chunk boundaries to minimize the chance that two concurrently-executing chunks attempt to communicate. Note that, at a hardware level, communication also includes name dependences (WAR and WAW) and false sharing. We neglect false sharing for now.

We propose to focus on the first communication operation in a code region where a thread may perform multiple communications with a second thread. We call such operation *Squash Hazard*. Typical Squash Hazards are synchronization operations and data races. Synchronizations involve passing information; data races may do so or may induce WAWs/WARs. Accesses to shared data protected by synchronization are typically not Squash Hazards. They are less likely to cause squashes because, by focusing on avoiding squashes on the synchronization, we mostly ensure that the data accesses do not conflict.

We propose ISA extensions that can be used to tailor the chunks around Squash Hazards (Table 1). First, the *Commit* operation finishes and commits the current chunk, and starts a new one. It gets translated into an instruction like *beginAtomic PC* from [1], which takes as argument the program counter (PC) of the safe version of the next chunk — to be used if the next chunk is repeatedly squashed. In this paper, we assume that, if no *Commit* operation is found after a certain, very large number of cycles, the hardware automatically commits the chunk and starts a new one.

The *Stall* operation stalls the processor for a period. In the *Check &Commit* and *Check&Stall* operations, the processor checks a certain condition and, based on the outcome, decides whether to commit the current chunk, or whether to stall, respectively. These two operations are likely implemented with a non-atomic combination of multiple instructions.

Operation	Functionality
Commit	Finishes and commits the current chunk, triggers a register checkpoint in hardware, and starts a new chunk.
Stall(Duration)	The processor stalls for a *Duration* number of cycles.
Check&Commit(Condition)	The processor checks *Condition* and, if it is true, it *Commits* the current chunk.
Check&Stall(Condition,Duration)	The processor checks *Condition* and, if it is true, it *Stalls* for *Duration* cycles.

Table 1: Operations to tailor chunks around Squash Hazards.

We consider five types of Squash Hazards, namely barriers, high-contention critical sections, middle-contention critical sections, flag set-wait, and data races. Next, we use the operations in Table 1 to tailor squash-elimination algorithms to these hazards.

4.2 Barriers

Figure 2(a) shows the code of a barrier, which we divide into an *Update* and a *Hold* section. Every thread arriving at the barrier updates *lock* and *count*. For the barrier to complete, each thread must observe the updates of the earlier threads. This requires that each thread necessarily commit its chunk sometime after its Update section. This can be attained by relying on the hardware to automatically commit a chunk after the thread has been spinning in the Hold section for a long time, as discussed in Section 4.1. However, our goal is to reduce the execution time and number of squashes by inserting instructions to tailor the chunks. In the following discussion, the ideas apply equally to lazy and eager conflict-detection architectures.

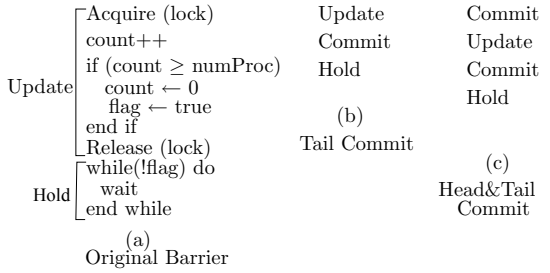


Figure 2: Squash-elimination algorithms for barriers.

Depending on whether the threads reaching the barrier are load-balanced or not, we propose two algorithms. If the threads are not load-balanced, they do not execute their Update sections concurrently. Consequently, we simply insert a Commit after the Update (Figure 2(b)). This reduces the chance of losing to squashes the work done by the thread before reaching the barrier and during the Update section. We call this algorithm *Tail Commit*.

If the threads are load-balanced, they bunch-up during the execution of the Update section and, therefore, can conflict there. Consequently, we insert Commits before and after the Update (Figure 2(c)). The first Commit reduces the chance of losing the work performed before the barrier. The second Commit reduces the time that a thread is vulnerable to conflicts, therefore minimizing the chance of losing the work in the Update section. We call this algorithm *Head&Tail Commit*. It is less efficient than *Tail Commit* because it has two Commits.

In all cases, each thread spinning on *flag* gets squashed when the last thread reaches the barrier and updates *flag*. Fortunately, no useful work is wasted as the threads were simply spinning.

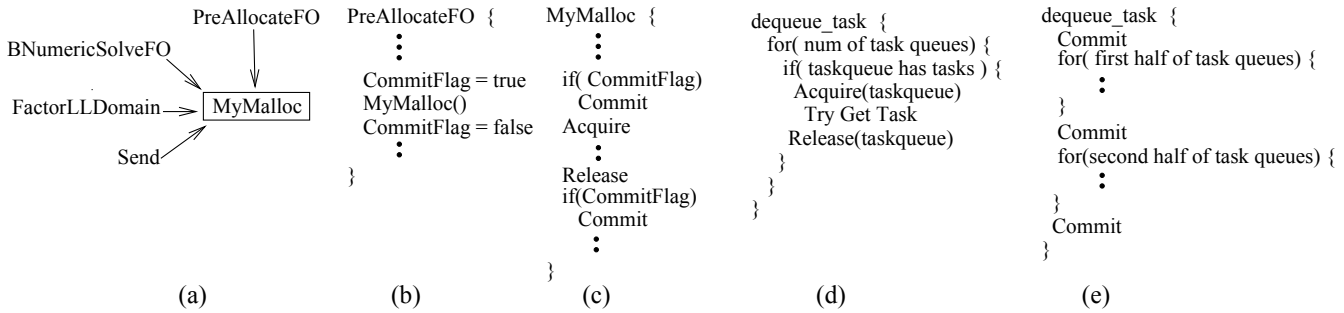


Figure 3: Squash-elimination algorithms for medium-contention critical sections.

4.3 High-Contention Critical Sections

In a high-contention critical section, multiple threads often attempt to acquire the lock concurrently. In a blocked-execution architecture with lazy conflict detection, multiple threads may proceed as if they all had acquired the lock. Eventually, all but one will be squashed. In an architecture with eager conflict detection, all threads may repeatedly be squashed.

We handle these Hazards with one of several algorithms. One involves inserting a Commit after the release (Figure 4(a)). This *Tail Commit* algorithm ensures that the release is immediately made visible to other threads. It also tries to prevent the squash of the work done by the thread before or during the critical section.

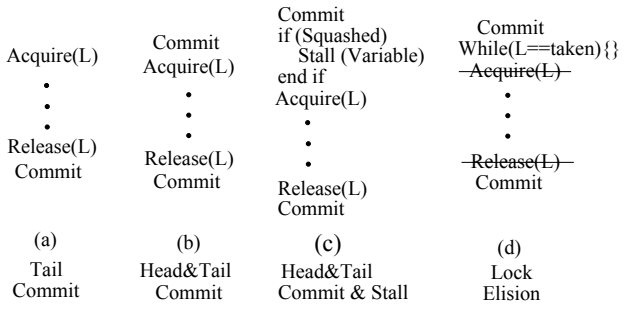


Figure 4: Algorithms for high-contention critical sections.

A second algorithm inserts a Commit immediately before the acquire and after the release (Figure 4(b)). This *Head&Tail Commit* algorithm further minimizes the chances of squashing the work performed before the critical section. However, it adds the overhead of an additional commit.

Another algorithm augments *Head&Tail Commit* with a stall for a certain number of cycles, if and when the chunk is squashed during critical section execution. This stall is performed before retrying the acquire (Figure 4(c)). We call this algorithm *Head&Tail Commit & Stall*. In a lazy architecture, when many threads execute the critical section concurrently, as soon as one thread commits, all the other threads get squashed. By stalling a variable number of cycles before restarting, we prevent lock-step squashes.

Finally, we can augment any of these algorithms with lock elision [19], where the acquire is replaced by a while loop with a plain load, and the release is removed. For example, Figure 4(d) shows the augmented *Head&Tail Commit* algorithm. Lock elision is used when the non-synchronization variables in the critical section are likely to be conflict-free. This algorithm can also be used in medium-contention critical sections.

4.4 Medium-Contention Critical Sections

In these critical sections, multiple threads occasionally try to enter concurrently. Most codes contain these critical sections, and they *often cause the majority of the squashes*. The algorithms for high-contention critical sections are suboptimal here because they lead to many nonessential commits and small chunks. Instead, we propose four different squash-removing algorithms.

4.4.1 Call-Path Commit

Often, we can accurately predict whether an instance of a medium-contention critical section will be squashed by examining the dynamic call path that lead to it. This is common in wrapper functions that have a critical section inside them. These functions are called from several different call sites but only a few of them consistently lead to squashes. Hence, we propose the *Call-Path Commit* algorithm, where the compiler forces chunk commits only if the critical section was invoked from a certain path.

An example is the malloc wrapper *MyMalloc* in *Cholesky*, which is called from multiple functions (Figure 3(a)). Its critical section is only squashed when called from *PreAllocateFO*. The compiler can place code in *MyMalloc* to commit only if it is called from *PreAllocateFO* (Figures 3(b)-(c)). Another implementation of this algorithm uses functional specialization for different call sites.

4.4.2 Loop Commit

Often, a loop accesses a critical section protected by a different lock in each iteration. While each individual lock has a low probability of contention, all the locks together have a high probability of inducing a conflict and a squash. Unfortunately, if the compiler places a commit after every single critical section, the result is very small chunks. Consequently, we propose the *Loop Commit* algorithm, where the compiler distributes the loop — i.e., it creates N loops, where each one iterates over 1/Nth of the original iterations — and introduces a commit after each loop. It also inserts a commit before the first loop starts.

This pattern occurs in helper functions that traverse data structures. Figure 3(d) shows *dequeue_task* from *Radiosity*, which traverses all the task queues, executing a critical section in each of them. Concurrently, another thread may operate on a queue and cause a squash. Figure 3(e) shows the code resulting from our algorithm, assuming that we distribute the loop into two.

4.4.3 Check&Stall Algorithm

A common sharing pattern involves two critical sections protected by the same lock variable that repeatedly communicate (e.g., in a producer-consumer manner). This is shown in Figure 5(a). During execution, one thread (*j*) has typically finished its critical section by the time the second one (*i*) attempts to start its own. However, sometimes, the second thread (*i*) arrives early, and tries to enter its critical section before thread *j* has exited its own (Figure 5(b)). The result is a conflict on at least the synchronization variable, and the eventual squash of one of the chunks.

To handle this pattern, we propose the *Check&Stall* algorithm.

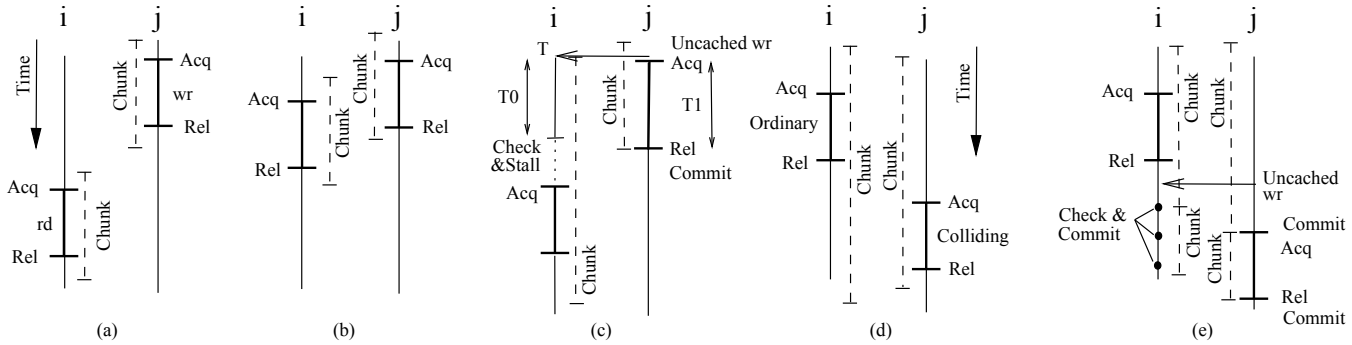


Figure 5: The *Check&Stall* and *Check&Commit* algorithms.

The idea is to precede thread *j*'s lock acquire by a compiler-inserted message to thread *i*. The message is used as a hint that *i* needs to wait before attempting an acquire, to allow thread *j* to complete its critical section and commit. A possible implementation of such a message and wait is an uncached write to memory by the processor running thread *j* and an uncached Check&Stall operation (Table 1) to the same location by the processor running thread *i*.

Figure 5(c) shows the operation of the *Check&Stall* algorithm. In the early thread (*j*), the compiler inserts an uncached write before the lock acquire and a commit after the release. The write stores the current time (*T* in the figure). In the other thread (*i*), the compiler inserts an uncached Check&Stall immediately before the lock acquire. With this primitive, thread *i* can stall for a time that is larger than the expected duration of the critical section in thread *j* (*T1* in the figure) minus the time elapsed since the uncached write (*T0* in the figure). By the time this stall is over, thread *j* will have committed its critical section accesses, and thread *i* will be capable to acquire the lock without conflicts.

This algorithm applies beyond a pair of repeatedly-communicating processors. It can support migratory sharing, where the pairs of communicating processors continuously change. It can also support the presence of multiple, changing consumer processors.

4.4.4 Check&Commit Algorithm

Another common pattern also involves two critical sections in different threads protected by the same lock. The first critical section (*Ordinary*), is executed relatively frequently by different processors and is not frequently squashed. The second one (*Colliding*), is executed less frequently but, when it is, it typically collides with an Ordinary section and induces a squash. Figure 5(d) shows an example where thread *i* executes the Ordinary section and, before *i* commits the chunk, thread *j* executes the Colliding section and induces a squash.

To handle this pattern, we propose the *Check&Commit* algorithm. The idea is for a thread (*j* in the example) to issue a hint when it is at a certain distance of starting to execute the Colliding section. This hint indicates that any thread that has executed an Ordinary section (*i* in the example) should commit its chunk immediately, to avoid a squash. This hint and commit can be implemented with an uncached write to memory by the processor running thread *j* and an uncached Check&Commit operation (Table 1) to the same location by the processor running thread *i*.

Figure 5(e) shows the *Check&Commit* algorithm. As the Colliding section in thread *j* is highly contended, it is surrounded by Commits. Moreover, the compiler inserts an uncached write in *j* much before *j* reaches the Colliding acquire. In thread *i*, the compiler inserts an uncached Check&Commit at periodic intervals after the Ordinary section. This operation checks whether the uncached write has been performed and, if so, commits the current chunk. Once the commit is performed, the remaining Check&Commits have no effect (Section 5.3).

The compiler has to carefully select where the Check&Commit operations are placed. Since each of them may mark the boundary of two atomic chunks, the compiler may be unable to optimize code across Check&Commit operations.

4.5 Flag Set-Wait

A flag Set and its corresponding Wait form a common Squash Hazard. In general, a squash occurs if the Wait executes before the chunk with the Set commits. In our algorithms, if a Set-Wait causes a high squash rate, we insert a commit after the Set. This ensures that the Set is observed as soon as possible. In addition, depending on the type of Wait present, we apply one of the following three algorithms.

If the Wait involves no work beyond checking the flag, we propose the *Head Commit* algorithm of Figure 6(a). The algorithm first checks the flag and, if it is not set, commits the chunk and spins on the flag until set. By checking the flag before committing, we avoid the overhead of a commit if the flag is already set. However, if the flag is not yet set, we commit to protect the work before the Wait.

```
if(!flag){
  Commit
  While(!flag) { }
}
```
(a)

```
if(!flag){
  Commit
  While(!flag) {
    Work
    Commit
  }
}
```
(b)

```
if(!flag){
  Commit
  iter=0
  While(!flag) {
    iter++
    Work
    if(iter MOD Num == 0)
      Commit
  }
}
```
(c)

Figure 6: Squash-elimination algorithms for Wait.

If the Wait is in a loop that performs a large amount of work between consecutive checks of the flag, we use the *Head&Iteration Commit* algorithm of Figure 6(b). The algorithm first checks the flag. If it is not set, it commits the chunk and then spins on the flag until it is set. Each iteration of the spinloop performs the original per-iteration work followed by a commit. If there is an exit from the loop body (e.g., due to a goto or a break statement), then a commit should be placed at that exit as well to save the work done.

If the Wait is in a loop that performs a small amount of work between consecutive checks of the flag, we use the *Head&Loop Commit* algorithm of Figure 6(c). The algorithm is the same as the *Head&Iteration Commit* except for the following. To reduce the

overall commit overhead, we do not commit at every iteration of the loop. Instead, we commit only once every few iterations (*Num*). This pattern is found in some fuzzy-barrier implementations.

4.6 Data Races and Library Functions

Typically, a data race is a Squash Hazard. A data race between two concurrently executing chunks causes a squash. Unless a race-detection tool is available, it is hard to pinpoint the racing references. Consequently, if synchronization is not the reason why chunks are getting squashed, the compiler decreases the size of the conflicting chunks by placing one or more commits inside them. The result may be that the squash disappears because the racing chunks do not overlap, or that less work is squashed.

Library functions are handled like the rest of the code by our algorithms. Some frequently-used ones like malloc, free, setlocal, or signal, use locks and/or other global shared state and, therefore, cause squashes. These library functions should be modified to work on thread-private data as much as possible. For example, malloc should work first on a private pool of memory before accessing a global pool, as in TCmalloc [10].

4.7 Summary

Table 2 summarizes the squash-elimination algorithms proposed for each type of Squash Hazard.

Squash-Elimination Algorithms	Squash Hazards				
	Barrier	High-Cont Critical Section	Med-Cont Critical Section	Flag Set-Wait	Data Race
Tail Commit	X	X			
Head&Tail Commit	X	X			
Head&Tail Commit&Stall		X			
Lock Elision		X	X		
Call-Path Commit			X		
Loop Commit			X		
Check&Stall			X		
Check&Commit			X		
Head Commit				X	
Head&Iteration Commit				X	
Head&Loop Commit				X	
Decrease Chunk Size					X

Table 2: Squash-elimination algorithms for Squash Hazards.

5. IMPLEMENTATION

While a static compiler can identify the synchronization points in a program, it can hardly assess their level of contention. Consequently, we envision our algorithms to form a framework inside a dynamic compiler. We call the framework *FlexBulk*. It is composed of three parts: *Profiler*, *Analyzer*, and *Optimizer*. The profiler is a pass that annotates the code to record dynamic information related to squashes and commits. The analyzer uses the generated statistics to characterize the Squash Hazards and deduce the best set of squash-elimination algorithms for them. Finally, the optimizer transforms the code accordingly.

5.1 Profile Module

The profiler is a compiler pass that identifies Squash Hazards in the code and inserts instrumentation around them. As we execute the application, the instrumentation dumps relevant statistics. The profiler gives one ID to each Squash Hazard, irrespective of the addresses it accesses. For example, an acquire at a given PC gets one ID, although it may access different variables in different invocations. The profiler also keeps the recent procedure call history so that, when it reaches a Squash Hazard, it knows the context.

As the program executes, the instrumentation generates a software table where each row corresponds to one Squash Hazard, given by an ID and a call history context. A row keeps the type of Hazard (e.g., barrier) and counts of how many times it executed and was then squashed (*NumSquash*), or executed and then committed (*NumCommit*). The profiler also generates other information that enables the analyzer to characterize the Squash Hazards. For example, it generates the number of instructions executed between successive accesses to the same Hazard (*Instruction Gap*). To detect candidates for the *Check&Stall* and *Check&Commit* algorithms, the profiler causes the program to issue uncached writes as per Section 4.4.

5.2 Analysis Module

The analyzer uses the statistics generated by the profiler to choose the best squash-elimination algorithm for each Squash Hazard. For this, it combines the per-thread statistics into application-wide, and then uses the thresholds of Table 3 to decide what algorithm to use.

Threshold	Meaning
Th_{sq}	Minimum fraction of execution time wasted by a Squash Hazard to squashes to make it interesting
Th_{bar}	Minimum squash rate in the Update section of a barrier to qualify as load-balanced
$Th_{cs}^{high}, Th_{cs}^{med}$	Minimum squash rate in a critical section to qualify as high or medium-contention, respectively
Th_{sw}	Minimum squash rate in a flag set-wait to qualify as interesting
Th_{igap}	Maximum Instruction Gap between successive accesses to the same Squash Hazard to qualify for the Loop Commit or Head&Loop Commit algorithms

Table 3: Thesholds used by the analyzer.

As shown in Table 3, if a Squash Hazard causes less than Th_{sq} squash cycles (as a fraction of the total application execution time), it is not considered. Otherwise, the analyzer computes the *Squash Rate* of the Hazard as $\frac{NumSquashes}{NumSquashes+NumCommits}$, where NumSquashes and NumCommits are the number of times the chunk with the Hazard is squashed and committed, respectively. For barriers, the squash rate is compared to Th_{bar} to determine if the barrier is load-balanced. Depending on the outcome, the analyzer chooses the algorithms of Figures 2(b) or (c).

For critical sections, the squash rate is compared to Th_{cs}^{high} and Th_{cs}^{med} to see if the critical section is high-contention, medium-contention, or otherwise. If it is high- or medium-contention, and the non-synchronization variables are likely to be conflict-free, the analyzer chooses the lock elision algorithm of Figure 4(d).

Otherwise, the analyzer proceeds as follows. For high-contention critical sections, it chooses the *Head&Tail Commit & Stall* algorithm of Figure 4(c). The stall is set as follows. We assume that half of the processors (*P/2*) contend in the acquire. Therefore, each processor gets a random number from 0 to *P/2-1* and multiplies it by *L*, where *L* is the expected duration of the critical section. This is the stall time of the processor.

For medium-contention critical sections, the analyzer considers the four algorithms of Section 4.4 in the following priority order: *Loop*, *Call-Path*, *Check&Commit* and *Check&Stall*. As soon as one is applicable, it is chosen, and further examination is terminated. The *Loop Commit* algorithm is applicable if the Instruction Gap

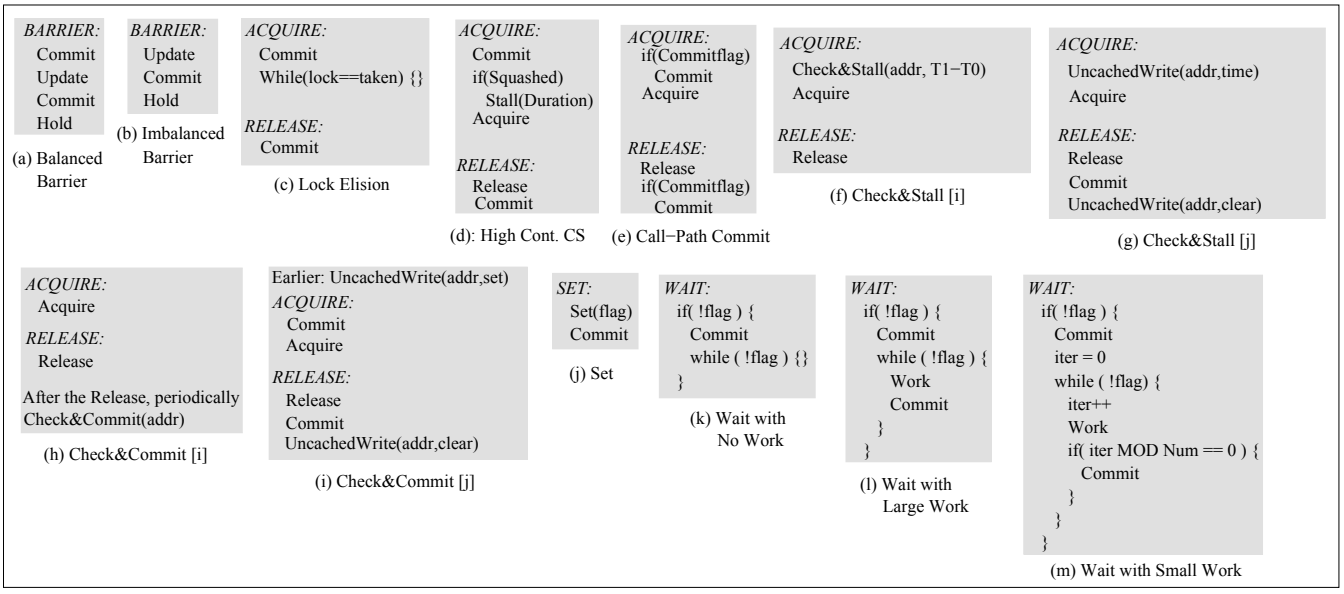


Figure 7: Synchronization macros used by the optimizer.

between successive accesses to the Hazard is less than Th_{igap}. In this case, the analyzer distributes the loop and inserts commits as per Figure 3(e). The *Call-Path Commit* is applicable if there is one or more call-path contexts for which the squash rate is higher than Th_{cs}^{high}. In this case, the analyzer uses the code of Figure 3(c). The *Check&Commit* and *Check&Stall* are applicable if, after the profiler instruments the code appropriately and measures the squash rate, the latter is higher than Th_{cs}^{high}. More details on the implementation of these two algorithms are presented in Section 5.3.

For a flag set-wait, if its squash rate is less than Th_{sw}, it is not considered further. Otherwise, the set is followed by a commit. Moreover, if the wait has no work, the analyzer chooses the *Head Commit* algorithm of Figure 6(a). If it has work, the analyzer compares the Instruction Gap between successive accesses to Th_{igap}. Based on this, it chooses the *Head&Iteration* or *Head&Loop Commit* algorithms of Figures 6(b) and (c).

5.3 Optimization Module

The optimizer transforms the code around the Squash Hazards according to the chosen algorithms, and then recompiles it. The transformations use the four operations of Table 1, and augment high-level synchronization constructs such as M4 macros [14]. In the following discussion, we assume we have macros for BARRIER, ACQUIRE, RELEASE, SET, and WAIT.

The BARRIER macro is shown in Figures 7(a) or (b) depending on whether the barrier is load-balanced or not. The Update and Hold sections were shown in Figure 2(a).

The ACQUIRE and RELEASE macros for critical sections that use lock elision are shown in Figure 7(c). Otherwise, those for high-contention critical sections and for the *Call-Path Commit* algorithm are shown in Figures 7(d) and (e), respectively. For the *Loop Commit* algorithm, the macros do not change.

The ACQUIRE and RELEASE macros for the two thread types in the *Check&Stall* algorithm are shown in Figures 7(f) and (g). Those for the *Check&Commit* algorithm are in Figures 7(h) and (i). The thread that initiates the communication (call it *j*) performs an uncached write to a memory location (call it *addr*). In the *Check&Stall* algorithm, it writes the current time; in the *Check& Commit* one, it sets the location. Such write does not terminate the current chunk. Moreover, if *j* gets squashed, it is acceptable to leave the *addr* location set. Later, in both algorithms, when thread *j* commits its chunk, it clears *addr*, so that future reader threads do not get confused.

The reader threads (call them *i*) perform either Check&Stall or Check&Commit operations in the *Check&Stall* and *Check&Commit* algorithms, respectively. Recall from Section 4.1 that each of these operations is implemented as a non-atomic combination of multiple instructions. They include an uncached read to *addr* that does not terminate the chunk. In the *Check&Commit* algorithm, once thread *i* commits the chunk, it sets a local variable, so that the execution of any future Check&Commit operation has no effect until *i* finds another commit operation in its execution.

If a Set-Wait pair has a high squash rate, we use the SET macro of Figure 7(j) and one of the WAIT macros of Figures 7(k)-(m). For library functions, we use the same algorithms as for the rest of the code.

6. EVALUATION SETUP

We model the FlexBulk profiler with an analysis tool based on the Pin [13] binary instrumenter. The tool runs the applications with a smaller input data set (training set) and dumps commit and squash information. The FlexBulk analyzer and optimizer are modeled with a set of automated scripts that analyze the information, and then choose the appropriate squash-elimination algorithms and M4 macros. The resulting applications are then run with a bigger input data set (deployment set) under Pin.

Pin is connected to a detailed multiprocessor architecture simulator based on SESC [20]. We model a chip multiprocessor (CMP) with 16 cores interconnected with a multistage network. The architectural parameters are shown in Figure 8(a). We simulate an architecture similar to BulkSC [6]. This is a lazy conflict-detection system with potentially false positive squashes due to signatures. The signatures are similar to Notary's PBX [24]. Figure 8(a) shows representative values of the visible latency of a commit operation (400-500 cycles) and a squash operation (1500-4000 cycles beyond the work wasted). The commit latency includes compressing the signatures, sending them to the arbiter and, when the latter responds, clearing them. The squash latency includes using the signatures to identify and invalidate the dirty speculative lines in the caches and

Simulated System Configuration
System: CMP with 16 cores Core: two−issue, in−order at 1GHz
L1: 16KB, 4−way, 32B line Private, write−through, 16−entry MSHR Hit delay: 2 cycles round trip L2: 256KB, 8−way, 32B line Private, write−back, 16−entry MSHR Hit delay: 8 cycles round trip Victim cache: 64−entry, full−asso, 32B line Directory cache module per core: 32K entries, 8−way, full map Multistage Interconnect L2 miss delay: To other L2s: 30 cyc round trip (avg) To memory: 250 cycles round trip Mem: DDR2 DRAM. Datarate: 667MHz
Write/read signature: 4096 bits each similar to Notary's PBX Commit cost: approx. 400 − 500 cycles Squash cost: approx. 1500 − 4000 cycles Target chunk size: 2K, 5K, 10K, 20K ins

(a)

Application	Training Input Size	Deployment Input Size
PARSEC		
Canneal	simsmall	simmedium
Ferret	simsmall	simmedium
Streamcluster	simsmall	simmedium
Swaptions	simsmall	simmedium
SPLASH−2		
Barnes	4K particles	32K particles
Cholesky	tk15.O	tk29.O
Ocean	66 * 66 ocean	258 * 258 ocean
Radiosity	test	room
Radix	64K keys	256K keys
Raytrace	teapot	car

(b)

Envir.	Explanation
PerfBase	**Perfect.** No cache overflow. Perfect disambiguation of addresses for conflicts (addresses are at word level and access information is in cache lines)
PerfOpt	*PerfBase* after our optimizations
FSBase	**False Sharing.** Cache overflow is possible. False sharing is possible (disambiguation uses cache−line addresses). Access information is in cache lines
FSOpt	*FSBase* after our optimizations
RealBase	**Real.** Cache overflow is possible. False sharing is possible (disamb. uses cache−line addresses). False positives are possible (access inform. is in signatures)
RealOpt	*RealBase* after our optimizations

(c)

Th_{sq}	Th_{bar}	Th_{cs}^{high}	Th_{cs}^{med}	Th_{sw}	Th_{igap}
1%	60%	75%	30%	30%	400 instr.

(d)

Figure 8: Parameters used in the evaluation: simulated system configuration (a), applications and input sizes (b), environments considered (c), and threshold values used (d). We use StrCl for Streamcluster.

then restoring a checkpoint. The squash latency varies significantly based on the cache state.

The *target* chunk sizes evaluated are 2K, 5K, 10K, and 20K instructions. These are sizes *targeted* by the software; the effective average sizes are smaller because our algorithms have limited information and need to be conservative. Moreover, they insert early commits to avoid squashes. For example, with the 20K target, we attain an average chunk size of 17,000 instructions. We use 2K as *baseline* because it is used in [6].

We use all the SPLASH-2 and PARSEC applications that have a squash overhead of 3% or more for the 20K target chunk size. Figure 8(b) shows the applications and input sets.

We model the environments of Figure 8(c). There are three main groups: *Perf*, *FS*, and *Real*. *Perf* is a perfect environment, where chunk squashes only occur due to *same-address* data conflicts (memory addresses are examined at word granularity and the cache tags record which locations were accessed). *FS* is *Perf* augmented with squashes caused by cache overflows and conflicts due to false sharing (addresses are examined at line granularity). Given the sizes of our caches and victim caches, cache overflow is very minor. *Real* is *FS* augmented with squashes caused by false positive conflicts due to address aliasing in signatures. In each of these three groups, we have an environment without our optimizations (*Base*) and one with them (*Opt*). When we refer to an environment, we append the target chunk size that it uses in instructions (e.g., *PerfBase2K*). The default is 2K chunks. Figure 8(d) shows the thresholds used by the analyzer.

7. EVALUATION

We first characterize the *PerfBase* system, then examine the improvement attained by our framework (*PerfOpt*), and then measure the realistic systems with overflow, false sharing and false positives (*FSOpt* and *RealOpt*).

7.1 PerfBase: Performance Characterization

Figure 9 compares the execution time of *PerfBase2K*, *PerfBase5K*, *PerfBase10K*, and *PerfBase20K*. The bars are normalized to *PerfBase2K*, as in BulkSC. They are broken down into *Squash* (squashed work plus squash overhead), *Commit* (commit overhead), and *Useful* (the rest). We also show the geometric mean bars. Note that the geometric mean bars cannot be meaningfully broken down into the three components.

Under this perfect, yet unoptimized environment, *PerfBase2K* suffers moderately from squashes. Because of its small chunks, it also suffers from commit overhead. As the target chunk size increases, squash time generally goes up and commit time decreases. For 20K chunks, the commit time is minor, but the squash time is significant, especially for some applications such as Swaptions, Ocean and Radiosity. Based on the geometric mean, *PerfBase20K*'s execution time is 10% higher than *PerfBase2K*'s, while the best design point is 5K chunks.

The average fraction of time wasted to squashes changes from 8% in *PerfBase2K* to 25% in *PerfBase20K*. Our goal is to understand and remove this overhead. From now on, we focus on the environment with 20K target chunks because it has the least commit time and, therefore, the highest performance potential. We use the environment with 2K target chunks as baseline.

7.2 PerfBase: Breakdown of Squash Cycles

Table 4 breaks down the squash time of *PerfBase20K* into the contribution of each Squash Hazard type. The numbers are given as percentages of the total execution time of the application. *Other* are unassigned squashes.

Appl.	Barrier	High Cont. Critic. Sect.	Medium Cont. Critic. Sect.	Flag Set Wait	Data Races	Other	Total Squash
Canneal	0.0	0.0	5.9	0.0	0.0	0.5	6.4
Ferret	0.0	0.0	0.0	0.0	2.4	0.7	3.1
StrCl	**19.4**	0.0	0.0	0.0	0.0	2.2	21.6
Swaptions	0.0	0.0	**49.5**	0.0	0.0	2.6	52.1
Barnes	0.0	4.6	0.0	0.5	0.6	0.1	5.8
Cholesky	0.1	0.3	**10.7**	0.8	0.0	1.4	13.3
Ocean	**33.9**	1.1	0.0	0.0	0.7	0.7	36.4
Radiosity	0.9	**13.0**	**41.5**	**23.3**	0.0	7.8	86.5
Radix	**10.4**	0.0	0.0	0.0	0.0	1.2	11.6
Raytrace	0.0	**10.2**	0.0	0.0	0.0	0.4	10.6
Average	6.5	2.9	10.8	2.5	0.4	1.9	24.7

Table 4: Squash time as a percentage of execution time in *PerfBase20K* for each type of Squash Hazard.

The largest contributors to squashes are medium-contention critical sections and barriers. The former strongly affect Swaptions

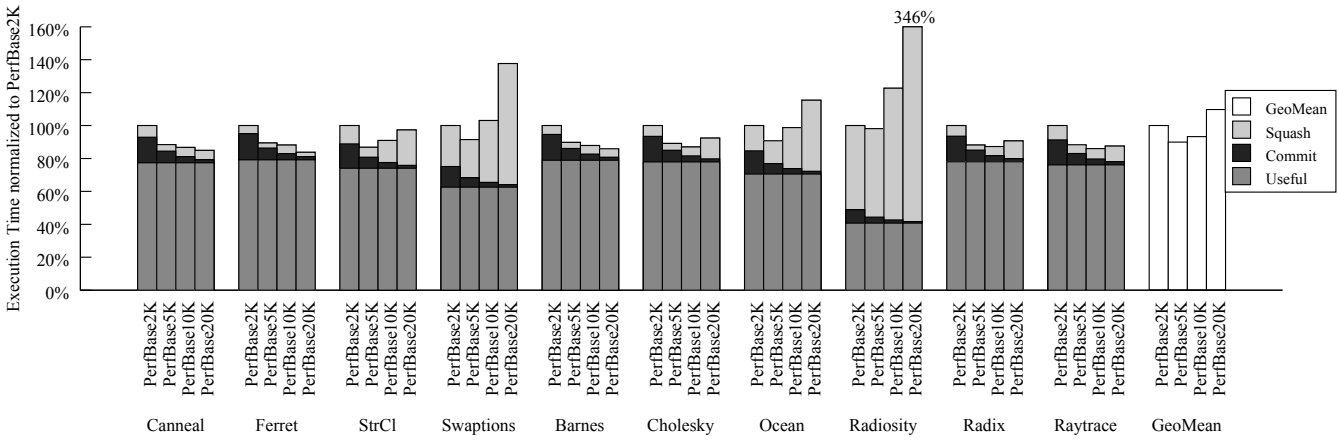


Figure 9: Execution time breakdown of *PerfBase* for different target chunk sizes.

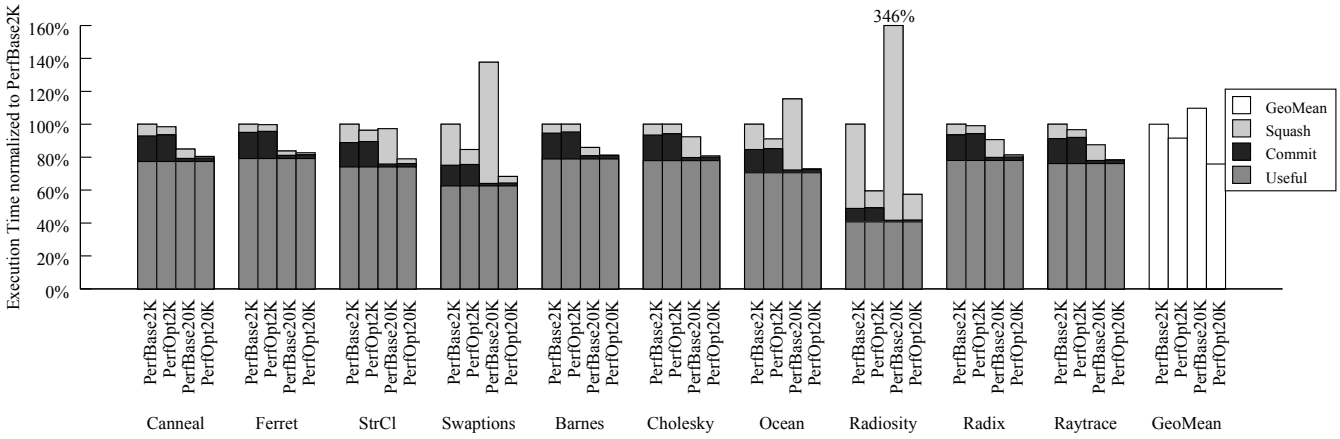


Figure 10: Impact of FlexBulk on the execution time under *Perf* for 2K and 20K target chunk sizes.

and Radiosity, while the latter impact StrCl and Ocean. Other categories are also noticeable. For example, squashes due to high-contention critical sections affect Radiosity and Raytrace, while flags have a major impact on Radiosity. Overall, we need a variety of algorithms to handle all of these cases.

7.3 PerfOpt: Performance Using FlexBulk

Figure 10 compares the execution time of two environments before optimization (*PerfBase2K* and *PerfBase20K*) and after (*PerfOpt2K* and *PerfOpt20K*). The bars are normalized to *PerfBase2K*.

FlexBulk's intelligent chunk generation eliminates much of the squash time in both environments. The reduction in the 20K chunk environment is large. Specifically, the average fraction of squash time in *PerfBase20K* was 25%, which now becomes 4% in *PerfOpt20K*. The reduction in the 2K chunk environment is relatively lower because our algorithm must necessarily be more conservative: cutting small chunks smaller risks increasing the commit time. Overall, as shown by the mean, *PerfOpt20K* is now 17% faster than *PerfOpt2K*. Neither has much squash time, but *PerfOpt20K* has less commit overhead thanks to using larger chunks.

To understand *PerfOpt20K*'s improvement, Table 5 shows the reduction in squash time delivered by each of our squash-elimination algorithms with 20K target chunks. The reduction is given as a percentage of the squash time in *PerfBase20K*. Each column refers to one algorithm and each row to one application. A given box in the table shows the reduction in squash time attained by applying *only* that algorithm. For example, in Radix, applying *Barrier Tail Commit* alone eliminates 74.8% of the squash time. The algorithms missing did not have any noticeable impact. The last column corresponds to the application of all the algorithms together.

On average, FlexBulk eliminates about 92% of the squash time in the applications. This is a large reduction. The algorithms that have the biggest impact vary across applications. Moreover, as shown in bold in the table, many of the algorithms in FlexBulk are needed. For example, the two flavors of barrier are effective. High-contention critical sections are optimized with *Head&Tail Commit & Stall* and with *Lock Elision*. The latter is effective for the *loadtree* function in Barnes, as illustrated in Section 3.

Some of the algorithms for medium-contention critical sections are key to one or more applications. For example, *Call-Path Commit* is effective for the *Get* wrapper function for atomic operation in Canneal. *Loop Commit* and *Check&Commit* are effective in Radiosity. The former is used in the *enqueue_task* function, which accesses a lock repeatedly in a loop; the latter algorithm is used in a critical section in the *dequeue_task* function, where a thread gets squashed occasionally, when other threads enqueue a task in its queue.

The *Head&Iteration Commit* algorithm for flags is also beneficial. Specifically, it is very effective in a large-work flag wait that is part of a fuzzy barrier in Radiosity. Finally, our algorithm for data races is useful in Ferret, which has races in the *emd* function.

7.4 FSOpt and RealOpt: Performance with Overflow, False Sharing & False Positives

The *FS* environment augments *Perf* with squashes due to false sharing and cache overflows, while *Real* augments *FS* with squashes

Appl.	Barrier		High-Cont. CS		Med-Cont. Crit. Sec.					Flag Set Wait		D. Race	All Together
	Tail Commit	Head & Tail Commit	Head & Tail Commit & Stall	Lock Elision	Call-Path Commit	Loop Commit	Check& Stall	Check& Commit	Lock Elision	Head Commit	Head & Iteration Commit	Decrease Chunk Size	
Canneal	0	0	0	0	**89.7**	0	0	0	0	0	0	0	89.7
Ferret	0	0	0	0	0	0	0	0	0	0	0	**69.5**	69.5
StrCl	**71.1**	15.4	0	0	0	0	0	0	0	0	0	0	87.4
Swaptions	0	0	0	0	**93.3**	0	0	0	0	0	0	0	93.3
Barnes	0	0	**74.1**	**83.3**	0	0	18.4	0	0	9.2	0	11.1	95.7
Cholesky	0	0	0	0	26.7	**58.9**	14.8	21.1	30.0	5.0	0	0	97.6
Ocean	10.8	**83.1**	2.4	0	0	0	0	0	0	0	0	2.4	99.0
Radiosity	0.3	0	17.8	0	0	**70.6**	20.1	**68.6**	0	0	**40.2**	0	96.3
Radix	**74.8**	20.5	0	0	0	0	0	0	0	0	0	0	91.1
Raytrace	0	0	**98.2**	0	0	0	0	0	0	0	0	0	98.2
Average	15.7	11.9	19.2	8.3	21.0	13.0	5.3	9.0	3.0	1.4	4.0	8.3	91.8

Table 5: Reduction in squash time delivered by each of our squash-elimination algorithms for 20K target chunks. The reduction is given as a percentage of the squash time in *PerfBase20K*.

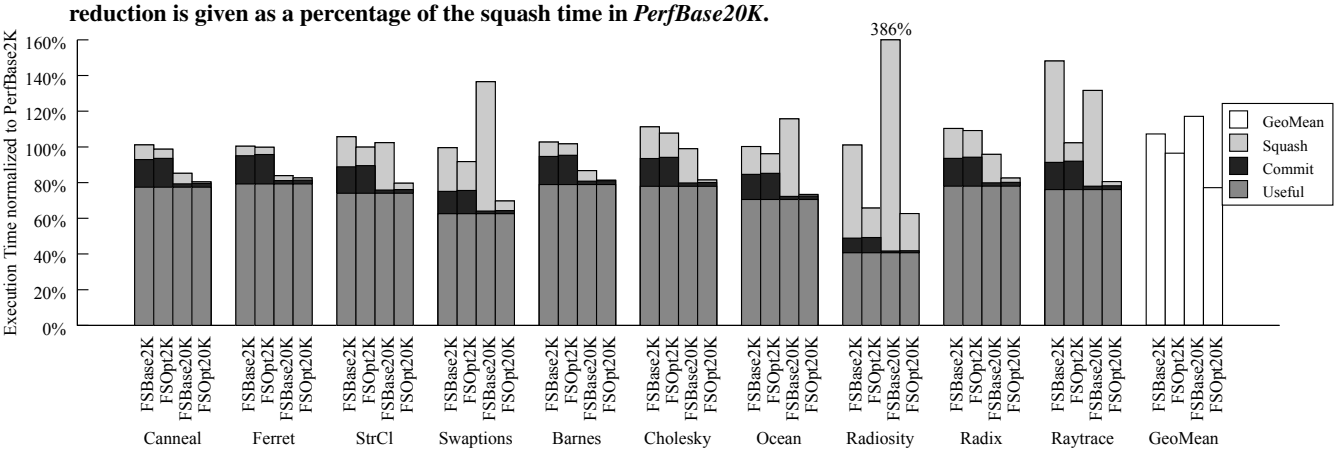


Figure 11: Impact of FlexBulk on the execution time under *FS* for 2K and 20K target chunk sizes. The bars are normalized to *PerfBase2K*.

due to false positives in the signatures. These environments are evaluated in Figures 11 and 12, which show the impact of FlexBulk on the execution time under *FS* and *Real*, respectively. Specifically, Figure 11 compares the 2K and 20K chunk environments under *FS* before optimization (*FSBase2K* and *FSBase20K*) and after (*FSOpt2K* and *FSOpt20K*); Figure 12 compares the 2K and 20K chunk environments under *Real* before optimization (*RealBase2K* and *RealBase20K*) and after (*RealOpt2K* and *RealOpt20K*). In both figures, the bars are normalized to the perfect environment with 2K target chunks (*PerfBase2K*).

As we move from Figure 10 (*Perf*) to Figure 11 (*FS*), and to Figure 12 (*Real*), the execution time of the unoptimized environments (*Base2K* and *Base20K*) increases because there are more squashes. For example, in *Base20K*, the squash time accounts for an average of 25%, 29%, and 40% of the time under *Perf*, *FS*, and *Real*, respectively. Most of the effect of false sharing appears as data-race induced squashes. They are common in functions like *malloc*, *free*, and their wrapper functions. On the other hand, the squashes due to false positives in the signatures are more random and distributed. Some applications such as Cholesky and Radix (Figure 12) suffer substantially from them.

Figures 11 and 12 also show that FlexBulk optimizes away much of the squash time in both environments. This can be seen in bars *FSOpt20K* (Figure 11) and *RealOpt20K* (Figure 12). Since both squash and commit time are low, FlexBulk delivers good speedups. From the mean, *FSOpt20K* is now 20% faster than *FSOpt2K*, and *RealOpt20K* is also 20% faster than *RealOpt2K*. This shows that our approach is robust and widely applicable.

7.5 Sensitivity to the Target Chunk Size

To understand the sensitivity to the size of the target chunks, we now compare the impact of FlexBulk on the execution time under 2K, 5K, 10K, and 20K target chunk sizes. Due to space limitations, we only show the geometric mean of the applications. The resulting execution times for the *Perf*, *FS*, and *Real* environments are shown in Figures 13(a), (b), and (c), respectively. In all of the charts, the execution time is normalized to that of the unoptimized perfect environment with 2K target chunks (*PerfBase2K*).

These charts show several trends. Initially, without the FlexBulk optimization (*Base* environments, shown in black), 5K chunks are the sweet spot. Such chunks provide the best combination of modest commit and squash overheads. However, when FlexBulk is applied (*Opt* environments, shown in gray), 20K chunks are always best. The reason is that their commit overhead was low, and now the squash overhead has been largely removed. Finally, the execution time of the unoptimized environments (*Base*) is clearly non-competitive, compared to the optimized ones (*Opt*) — especially in the most realistic scenario (*Real*).

7.6 Overall Speedups

Putting it all together, Figure 14 plots the execution speedups of our optimized environments with 20K target chunks (*Opt20K*) over the unoptimized ones with 2K target chunks (*Base2K*). We show a chart for each of the *Perf*, *FS*, and *Real* environments. In addition, each chart also shows the speedup that 20K would attain if we managed to eliminate all of the squash and commit overheads (*Ideal*). The charts show the geometric mean of all the codes.

Compared to unoptimized 2K target chunks as in BulkSC, FlexBulk

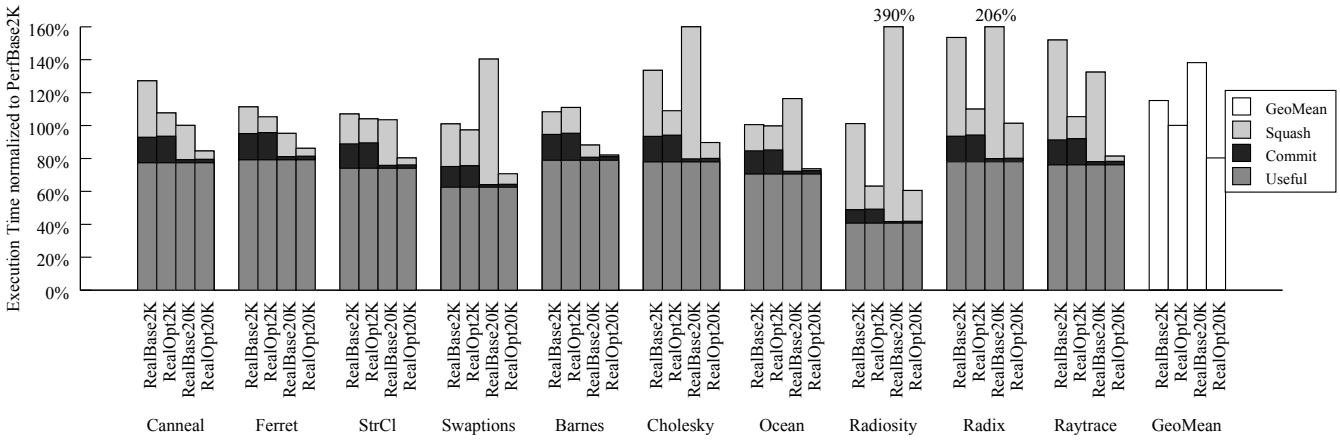
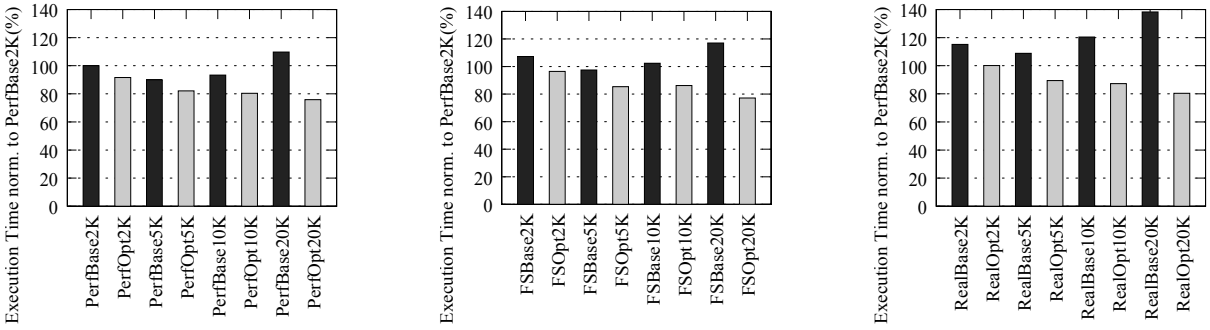
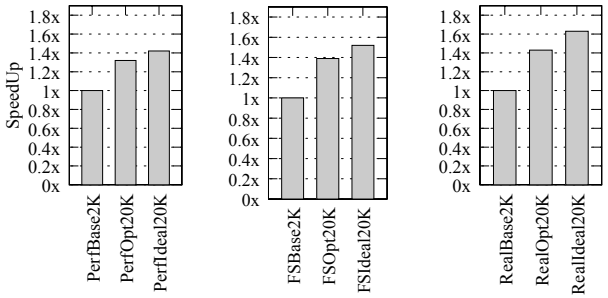


Figure 12: Impact of FlexBulk on the execution time under *Real* for 2K and 20K target chunk sizes. The bars are normalized to *PerfBase2K*.



(a): Perf (b): FS (c): Real

Figure 13: Impact of FlexBulk on the execution time under *Perf*, *FS*, and *Real* for 2K, 5K, 10K, and 20K target chunk sizes. The bars show the geometric mean of the applications and are normalized to *PerfBase2K*.



(a): Perf (b): FS (c): Real

Figure 14: Geometric mean speedup attained by FlexBulk and 20K chunks under the *Perf*, *FS*, and *Real* environments.

applied to 20K target chunks attains an average application speedup of 1.32x, 1.40x, and 1.43x for the *Perf*, *FS*, and *Real* environments, respectively. These are substantial speedups for the 16-threaded applications — especially given that we attain them with *very little* additional hardware support. Indeed, FlexBulk profiles and optimizes the code *in software*. Of all the operations in Table 1, the only one that really needs a new hardware instruction is Commit. However, even such an instruction only provides a software interface to already existing hardware mechanisms for chunk start and commit in the baseline blocked-execution architecture.

If we could magically eliminate all of the squash and commit overhead (*Ideal* bars), we would attain only modestly higher speedups, namely, 1.42x, 1.52x, and 1.62x, respectively. Consequently, our optimizations represent a good design point.

8. RELATED WORK

Several authors have proposed compiler-based techniques to optimize the execution of atomic chunks of instructions. Some of the most related efforts are those of Ahn *et al* [1], Neelakantam *et al* [17], and Borin *et al* [5, 4]. We described the first two in Section 2. They want the compiler to improve the code within chunks, either by optimizing for a certain control path, or by optimizing code across synchronization operations. Our goal, instead, is to enable large chunks in the first place, by starting-off with large chunks and intelligently setting the boundaries to avoid squashes. The efforts are synergistic in that our larger chunks will enable more aggressive compiler optimizations. Borin *et al* [5, 4] also try to attain larger chunks but, rather than focusing on squashes due to data conflicts, they focus on those due to cache overflows. They minimize the impact of limited cache capacity by devising novel hardware for two-level buffering [5] and conditional commit [4]. With our large caches and R signature use, we have not observed significant cache overflow. However, their techniques are complementary.

Bobba *et al* [3] present several high-level thread interaction patterns that cause thread stalls or squashes in transactional memory systems. They use contention management methods to remove

these pathologies. Our goal and approach are different. Our goal is to remove squash time, and do it by cutting the code into chunks at key points. They cannot cut transactions short.

Other authors have proposed hardware-based techniques for dynamic prediction and synchronization of cross-thread dependences for thread-level speculation (e.g., [8, 16]). These techniques can augment our FlexBulk framework.

9. CONCLUSION

A limitation of blocked-execution multiprocessors is that, if they use large chunks to minimize chunk-commit overhead and to enable aggressive compiler optimization, inter-thread data conflicts lead to frequent squashes.

To solve this problem, this paper has presented automatic techniques to form chunks that minimize the cycles lost to squashes. We identified and characterized common types of *Squash Hazards*. We then proposed squash-removing algorithms tailored to them. These are simple code transformations, often embedded in synchronization macros, that create chunk boundaries for minimal squashes. We described the FlexBulk software framework that chooses and applies these transformations. Finally, we evaluated FlexBulk on a simulated 16-processor blocked-execution architecture running PARSEC and SPLASH-2 codes. We showed that this approach is very effective. With 17,000-instruction chunks, FlexBulk eliminates, on average, over 90% of the squash cycles in the applications. As a result, compared to a baseline execution with fixed 2,000-instruction chunks as in BulkSC, the applications run on average 1.43x faster. Our next step is to exploit these larger chunks with novel compiler optimization.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers and the I-ACOMA group members for their comments. This work was supported in part by NSF under grant CCF-1012759; Intel and Microsoft under the Universal Parallel Computing Research Center (UPCRC); Sun Microsystems under the UIUC OpenSPARC Center of Excellence; DARPA under UHPC Contract Number HR0011-10-3-0007; and DOE ASCR under Award Number DE-FC02-10ER2599.

11. REFERENCES

[1] W. Ahn, S. Qi, J. Lee, M. Nicolaides, X. Fang, J. Torrellas, D. Wong, and S. Midkiff. BulkCompiler: High-performance sequential consistency through cooperative compiler and hardware support. In *Int. Symp. on Microarch.*, Dec 2009.

[2] C. Blundell, M. Martin, and T. Wenisch. InvisiFence: Performance-transparent memory ordering in conventional multiprocessors. In *Int. Symp. on Comp. Arch.*, June 2009.

[3] J. Bobba, K. Moore, H. Volos, L. Yen, M. Hill, M. Swift, and D. Wood. Performance pathologies in hardware transactional memory. In *Int. Symp. on Comp. Arch.*, June 2007.

[4] E. Borin, Y. Wu, C. Wang, and M. Breternitz. LAR-CC: Large atomic regions with conditional commits. In *Int. Symp. on Code Gen. and Opt.*, April 2011.

[5] E. Borin, Y. Wu, C. Wang, W. Liu, M. Breternitz, S. Hu, E. Natanzon, S. Rotem, and R. Rosner. TAO: Two-level atomicity for dynamic binary optimizations. In *Int. Symp. on Code Gen. and Opt.*, April 2010.

[6] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Int. Symp. on Comp. Arch.*, June 2007.

[7] H. Chafi, J. Casper, B. Carlstrom, A. McDonald, C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *Int. Symp. on High Perf. Comp. Arch.*, Feb 2007.

[8] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Int. Symp. on High Perf. Comp. Arch.*, Feb 2002.

[9] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *Archit. Sup. for Prog. Lang. and Oper. Sys.*, March 2009.

[10] S. Ghemawat and P. Menage. TCMalloc: Thread-caching malloc. //goog-perftools.sourceforge.net/doc/tcmalloc.html.

[11] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Int. Symp. on Comp. Arch.*, June 2004.

[12] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *Int. Symp. on Comp. Arch.*, June 2008.

[13] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Prog. Lang. Design and Impl.*, June 2005.

[14] E. Lusk, J. Boyle, R. Butler, T. Disz, B. Glickfeld, R. Overbeek, J. Patterson, and R. Stevens. *Portable programs for parallel processors*. Nov 1988.

[15] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Int. Symp. on Comp. Arch.*, June 2008.

[16] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. In *Int. Symp. on Comp. Arch.*, June 1997.

[17] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *Inter. Symp. on Comp. Arch.*, June 2007.

[18] S. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and reliable communication for hardware transactional memory. In *Par. Arch. and Comp. Tech.*, Sep 2008.

[19] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Int. Symp. on Microarch.*, Dec 2001.

[20] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, Jan 2005. http://sesc.sourceforge.net.

[21] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic. The Bulk Multicore Architecture for improved programmability. *Communications of the ACM*, 52(12), 2009.

[22] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. E. Smith, and M. Valero. Implementing kilo-instruction multiprocessors. In *Int. Conf. on Pervasive Sys.*, July 2005.

[23] T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *Int. Symp. on Comp. Arch.*, June 2007.

[24] L. Yen, S. C. Draper, and M. D. Hill. Notary: Hardware techniques to enhance signatures. In *Int. Symp. on Microarch.*, Dec 2008.